Laser Patroll Final Report

Eric Krause, *Griddle Master, Breakfast Confectionery Guild,*
Josh Moles, *Waffle Stomper, Breakfast Confectionery Guild,*
and Erik Rhodes, *Eggtravagant Chef, Breakfast Confectionery Guild*

***Abstract*—This work discusses a benchmarking of four sort algorithms tested against each other for a Portland State University ECE 588 project. The four sorts tested were C++'s `std::sort`, Quicksort, Merge sort, and Bitonic sort. This was accomplished by building a benchmark framework in C++ and then building each of the described algorithms to compare each of them running a sort against the same set of data. We found that in a multi-threaded case, our merge sort was able to outperform C++'s `std::sort`. Bitonic and Quicksort did not outperform, but still showed decent gains as the number of threads increased. These results were ran with a set of unsigned integers across multiple threads and could be used to help determine the performance of these algorithms when run on a `std::vector`.**

***Index Terms*—sorting, Bitonic sort, Quicksort, Merge sort, class.**

I. Introduction

SORTING data is a task in computer science that is highly important because it is a very common operation. This report discusses the work done by Laser Patroll [sic] for a team project at Portland State University for the ECE 588 Advanced Computer Architecture II course taught in Fall 2013. The focus of this project was twofold:

- First, to either identify parallel sorting algorithms and implement them, or parallelize existing serial (single-threaded) algorithms.
- Second, benchmark each of the selected algorithms against each other. This involved sorting lists of different lengths using varying numbers of threads, while measuring the relative performance and behavior of the selected algorithms against the known, highly optimized C++ `std::sort`.

The four algorithms benchmarked and discussed in this report are the C++ `std::sort`, Quicksort [1] [2], Merge sort [3], and Bitonic sort [4]. In this report, we first present an overview of these sorting algorithms in Section II, then cover the C++ testing framework designed to check the sorted data and gather results in Section III, and finally present the results of the four algorithms in Section IV.

II. Sorting Algorithms

This section will discuss the four sorting algorithms run.

A. Basic Sort

Basic sort was implemented by simply passing the data array to `std::sort`. In no case was it executed in a parallel fashion.

B. Quick Sort

Quicksort is a partition-exchange sort that uses pivots to reorder elements. While its worst case is $O(n^2)$, Quicksort tends to perform faster than other sorts in the same performance class. Caches improve the sort time substantially, due to Quicksort's tendency to exhibit spatial locality. It can be sorted in place, allowing for less memory usage during the process.

Figure 1 shows how the sorting is done. First a pivot value is selected. Starting with the outermost locations, the numbers are compared against the pivot. After each side has found a

E. Krause is a master of the arts of pancake capture at the Guild.

J. Moles enjoys the art of waffle design.

E. Rhodes does not understand why eggs are not more common in everyone's daily life.

Manuscript completed for 10 December 2013.

Algorithm 1 QuickSort

```
Require: input array src
Require: output array dst
  procedure QUICKSORT($src$)
      if (LENGTH($src$) $\leq 1$) then
          return $src$                                  ▷ an array of 0 or 1 elements is sorted
      end if
      Select and remove a pivot element from $src$
      Create empty lists $less$ and $greater$
      for all $x$ in $src$ do
          if $x \leq pivot$ then append $x$ to $less$
          else append $x$ to $greater$
          end if
      end for
  return CONCATENATE(QUICKSORT($less$), $pivot$, QUICKSORT($greater$))   ▷ two recursive calls
  end procedure
```

number belonging to the other side, the numbers are swapped in place. This is continued until the indexes have switched sides, at which point Quicksort is called recursively.

Quicksort is primarily a sequential algorithm, so to take advantage of parallel threading, the data was divided into different sections by the number of threads available. This domain decomposition is seen in Figure 2. Once each thread has finished, they are merged back together and sorted sequentially.

C. Merge Sort

Merge sort is a divide-and-conquer sorting algorithm which uniquely exhibits $O(n\ Log(n))$ time complexity regardless of the ordering of its inputs. The merge algorithm shown in Algorithm 2 creates a larger sorted array from two smaller sorted subarrays and is pivotal to how merge sort works. The merge is where the majority of actual work is done when performing a merge sort.

As shown in Algorithm 3, the input array is first split in half until only single elements remain. Since an array consisting of a single element is inherently sorted, these single elements are then merged together to create increasingly large subarrays, until all subarrays have been merged together in a binary fashion to form a full-sorted result array.

Since splitting the source data is such a fundamental aspect of merge sort, we developed an elegant algorithm which leverages the inherent domain decomposition present in merge sort. Our method requires no communication between threads, uses no mutexes, and uses a total of only N barriers for N threads. Furthermore, this algorithm ensures that each thread has access to approximately $\frac{1}{N}$ of the work and if blocked, is only ever blocked by a single other thread.

Shown in Algorithm 4, Multithreaded Merge Sort (MMS) modifies the merge sort shown in Algorithm 3 to pass in the number of remaining threads. If this number is zero, MMS functions identically to standard `mergesort()`.

If the number of remaining threads is greater than zero however, it means that a new thread will be created to handle one of the two `mergesort()` calls that MMS will make. The remaining threads will be decremented by one to reflect the new thread that will be created. Finally, the number of remaining threads will be divided by two and distributed between the two MMS calls. Our algorithm always gives the new thread the left half of the input array, and always distributes the remainder of the thread count division to the right. We measured no change in performance with changes to this convention so it was preserved.

An example of how MMS would create threads if initially launched with 4 threads is illustrated in Figure 3. At the top, MMS receives 4 threads. A new thread will be created. The 3 remaining threads (R) are divided up amongst T_1, the new thread and the recursive call to MMS performed by T_0. As shown, this threading model ensures that every thread has equal access to work at the bottom of the tree once N threads are created. In this example, each thread is responsible for 2 of the 8 subarrays. Note that many high-level threads would be blocked for the majority of execution if each call to MMS started (and had to wait on) two threads.

D. Bitonic Sort

The bitonic sort is a sort algorithm primarily used for parallel execution. The complexity is consistently $O(n\ Log^2(n))$, regardless of the data received. This sort consists of two different methods. First, the random data must be sorted or "built" into one bitonic sequence. After this is done, the bitonic sort algorithm is applied and the data is fully sorted (for the sequential version).

A bitonic sequence consists of sorted data divided into two sections, one with an ascending order and one with a descending order [Figure 4]. Once this has been arranged, the sorting can take place. To sort the data, each element is compared with the corresponding element of the other half. One half stores the minimum of the compared elements, while the other half stores the maximum number. Because the bitonic sort uses a divide-and-conquer algorithm, the data is split in half and sorted this way until atomic elements are reached.

To build the initial bitonic sequence, the bitonic sort is applied to every pair of elements. It is important to note that if there are only two elements in a dataset, it is guaranteed to be a bitonic sequence. After this has occurred, each element of the pair is compared with another pair and are sorted down to a single element. Figure 5 shows how this process is repeated until one large bitonic sequence has been reached. It is important to note that this sort only works on a data size that is a power of 2. For this reason, any data of a different was padded to allow the same sorting algorithm to be supplied.

The bitonic sort uses the same algorithm as Quicksort for parallel thread management as shown in Figure 2.

III. Testing Framework

C++ was selected as the language to implement the testing framework and all the sorting algorithms because of the relative ease and ubiquity of tools to compile for the language. The framework had the following objectives for its design:

- A common path to benchmark each function to ensure timing is measured consistently.
- Re-usability of common tasks between the sort algorithms.
- Common call to a Sort function allow blind usage of sort only passing the number of threads and a vector of data.
- Ability to swap object that data for sorting was stored in easily.

These goals were accomplished through an object oriented set of classes. A common `SortCommon` class was the parent of all the sort algorithms and contained the means to benchmark an algorithm, generate an unsorted data vector, check if a given set of data is sorted or not, divide set of data using domain decomposition, merging, and padding (used by Bitonic). Each of the sort functions were free to do the sort by any reasonable means necessary. The requirements were that the only two inputs the Sort gets is a set of data and the number of threads it is allowed to divide the data up across. Figure 6 shows the layout of these classes and the overall framework.

The algorithm was to sort the data in place (or at least modify the pointer to the new set of data) so that the benchmark function can then validate that the data was sorted. All four of the algorithms did the sort in place (modified the passed in data directly) or created a "destination" data set and then swapped the pointer of the two at the end. Memory utilization can become an issue with sets of data this large so copying or making copies of large subsets of the data was not performed in the algorithms.

C++'s `std::vector` filled with `unsigned int` was used as the storage object and data type for each of the algorithms. A `std::vector` is an efficient, dynamically linked list that was selected over an array for the ability to have an easily growing and shrinking memory space (in case an algorithm selected to do an operation involving such a task). We also feel that the C++ compiler can do a better job

Algorithm 2 Generic Merge of two Sorted Arrays (A,B)

```
Require: A,B are sorted arrays
Require: length(dst) = length(A) + length(B)
  procedure MERGE($dst, A, B$)
      $i, j, k \leftarrow 0$                                          ▷ initialize array indicies
      while ($i$ within bounds of $A$) and ($j$ within bounds of $B$) do
          $dst[k] \leftarrow$ MIN($A[i], B[j]$)
          Increment index of array containing min element ($i$ or $j$)
          Increment $k$
      end while
      while ($i$ within bounds of $A$) do                             ▷ i still in bounds
          $dst[k] \leftarrow A[i]$
          Increment $i$
      end while
      while ($j$ within bounds of $B$) do                             ▷ j still in bounds
          $dst[k] \leftarrow A[j]$
          Increment $j$
      end while
  end procedure
```

Algorithm 3 MergeSort

```
Require: dst and src are arrays of equal length
Require: low and high are indices into A
1: procedure MERGESORT($dst, src, low, high$)
2:     if ($low < high$) then
3:         $pivot \leftarrow$ FLOOR($low + high$)$/2$
4:         MERGESORT($SortedLeft, A, low, pivot$)                     ▷ sort left half
5:         MERGESORT($SortedRight, A, pivot + 1, high$)               ▷ sort right half
6:         MERGE($dst, SortedLeft, SortedRight$)                      ▷ merge left & right
7:     end if
8: end procedure
```

optimizing the use of a `std::vector` in our code compared to the relatively manual task of memory management in an array.

The benchmark function prepared a given set of data to pass to the virtual `Sort` function implemented by each algorithm. The benchmark function also verified that the data was actually sorted when the `Sort` function returned. The function only ran the timer for a benchmark of an algorithm during the `Sort` call portion of the testing. This was done to ensure that any overhead tasks performed by the benchmark function itself does not add to the time a given sort runs.

Data could be generated in a few different patterns depending on the type of sort desired for benchmarking. The three options available were a set of data that is actually in order, a set of data that is in complete reverse, and a set of data that was completely random. The first two were used primarily for verification and testing of aspects of the algorithm. The third (random) set of data was used for all testing shown Section IV of the report.

The data was checked for sorting using a single-threaded function to look through the elements. The function simply started at the front of the data and moved through all of the elements. If it found an element that was greater than the previous element (in other words, not sorted), the function would multiply the time it took for the sort to run by -1 and print a warning stating that the data was not sorted.

A few other other utility functions were also in the `SortCommon` class. A function that would take the length of the data and how many threads to divide the work amongst would return a `std::vector` containing a `std::pair` of the minimum and maximum index for a given thread to perform the sorting on. Bitonic and Quicksort used this in the

Algorithm 4 Multithreaded Mergesort (MMS)

```
Require: dst and src are arrays of equal length
Require: low and high are indices into src
 1: procedure MMS($dst, src, low, high, threads\_remaining$)
 2:     if $threads\_remaining == 0$ then                          ▷ cannot create a new thread
 3:         ▷ perform basic recursive MergeSort
 4:     else                                        ▷ a thread may be created in this call to MMS
 5:         if $(low < high)$ then
 6:             $pivot \leftarrow$ FLOOR($low + high$)/2
 7:             $t\_left$ = FLOOR(($threads\_remaining - 1$)/2)
 8:             $t\_right$ = $t\_left$+(($threads\_remaining - 1$)%2)
 9:             [New_Thread]← MMS(SortedLeft,A,low,pivot, t_left)   ▷ a new thread will be spawned to sort left
10:             MMS($SortedRight, A, pivot + 1, high, t\_right$)   ▷ right will be sorted recursively by current thread
11:             WAIT(New_Thread)
12:             MERGE($dst, SortedLeft, SortedRight$)
13:         end if
14:     end if
15: end procedure
```

functional decomposition of the data, but merge used its own algorithm. For bitonic, the data and the number of threads had to be a power of two without modification. A padding and unpadding function is also added to pad zeros to the data in the event the user requests a sort on data that does not meet the power of two requirement for bitonic.

The code was also checked with valgrind to verify that there were minimal memory leaks. Listing 1 shows the output from valgrind. The listing shows that the code frees everything of concern (definitely lost).

GitHub was also used for revision control. This entire project is available for download at http://github.com/jmoles/laser-patroll/. Doxygen is also commented in-line to generate a set of HTML and PDF documentation for the project.

IV. Results

This section provides the results for the tests of the algorithms. The results will show the amount of time it took each algorithm to sort with a varying number of threads and constant data size. It is important to note that throughout all of these tests, anything labeled as "basic" is `std::sort` running in a **serial** fashion. We did not wrap any parallelization around `std::sort` so that each of the algorithms are evaluated against the sort that would likely be used in C++.

The results are in Figure 7 for a data set of size 134,217,728 elements and Figure 8 for a data set of size 512 elements. These tests aimed to see how each algorithm performs with an increasing number of threads both with a small set of data and a large set of data. Ideally, each algorithm should get faster as the number of threads it has available to perform work increases.

Looking first at the large data set, Merge sort follows the ideal situation quite well and starts to reach a limit in execution time as it gets to the end. Quicksort seems fairly flat over the execution regardless of the number of threads allocated to it. This could be a results of the domain decomposition model of splitting up the work and bringing it back together. As the number of threads increases, there is a final merge that only merges two sets of data at a time. This could result in this final merge occupying more time.

Bitonic is affected by the same merge issue; however, it has another factor slowing it down: It can only operate on input arrays that are of a length that is a power of 2. Our Bitonic algorithm pads the inputs data with zeros to increase its length until it is equal to a power of 2, and then removes this additional padding after it is sorted. Evidently, the amount of time required to pad and unpad the input data is nontrivial, as Bitonic shows an average trend of decreasing execution time, but is susceptible to the requirement that the data and threads must be a power of two. As a result, Bitonic performs the best on the situations where the number of threads is a power of two and begins to slow down as it increases from a power of two, peaking one value right before the next power of two.

V. Conclusion

We provided the description of the sort algorithms as well as the means to run each of them. In summary, a multi-threaded Merge sort is a worthy competitor to the C++ `std::sort`. Quicksort and Bitonic do not perform poorly, but do not quite reach the speeds of the Merge sort. These algorithms could potentially used some more optimization to further improve their speed down the road. Overall, this project provided a good background to sorting, parallel computing, and the building of algorithms for optimal use on a multicore architecture.

Acknowledgment

The authors would like to thank Dr. Alaa Alameldeen for the opportunity to work on this project as well as what we learned throughout the term to complete this course. We also would like to give thanks to the respective authors and contributers to the sort algorithms. We also would like to thank Github for hosting of this open source project. This project was also powered by Ninkasi Brewing Company Total Domination IPA and thanks goes out to them. Please remember to drink responsibly when sorting and sort responsibly when drinking.

References

[1] C. A. R. Hoare, "Algorithm 64: Quicksort," *Communications of the ACM*, vol. 4, no. 7, pp. 321–322, 1961. [Online]. Available: http://dx.doi.org.proxy.lib.pdx.edu/10.1145/366622.366644
[2] C. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, 1962. [Online]. Available: http://comjnl.oxfordjournals.org/content/5/1/10.short
[3] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.
[4] K. Batcher, "Sorting networks and their applications," *Proceedings of the April 30–May 2, 1968, spring joint ...*, no. Figure 1, pp. 307–314, 1968. [Online]. Available: http://dl.acm.org/citation.cfm?id=1468121
[5] Quicksort. [Online]. Available: http://www.algolist.net/Algorithms/Sorting/Quicksort
[6] J. Mellor-Crummey. (2012) Comp 322 lecture 28: Bitonic sort. [Online]. Available: https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s12-lec28-slides-JMC.pdf?version=1&modificationDate=1333163955158

Eric Krause Eric "Josh Moles" Krause was born with 2 arms and legs and becomes unbearably "hangry" if he does not routinely eat food. A connoisseur of socks, scarves, ties, and other clothing items with high length:width ratios. He is master of footsie, commonly practicing playing footsie with himself for hours on end (which he refers to as "spiderman-ing") but nearly always losing because so skill, wow, very talent. He attributes his success to the Johnny Jump Up given to him by his uncle Harold when he was 2.

After being diagnosed with a chronic psychological illness known as the meat sweats he decided to make the best of the situation, and he now regularly invites friends over for his signature "meat sweat" parties, where guests are assaulted with live fish and encouraged to participate in Tuvan throat singing sing-alongs. He believes in gravity.

His hobbies include mingling, being stubborn, and yelling. He also enjoys a good old-fashioned reverse-finagling from time to time, and he is constantly humbled by his incredible intelligence. Eric tries to convince everyone that his opinions mean nothing, but they never seem to care. In college, Krause found a passion for deleting others work, and this lead him to

His current website, sauerkrause.org, specializes in text$\rightarrow$text conversions of nearly any document format. It employs a patented document$\rightarrow$ image$\rightarrow$ scanner$\rightarrow$ text$\rightarrow$speech$\rightarrow$ calligraphy$\rightarrow$ Braille$\rightarrow$ wood$\rightarrow$ mimeograph$\rightarrow$ interpretive dance$\rightarrow$ speech-to-text algorithm, which preserves figuratively none of the original content.

Josh Moles Josh "Badness 10,000" Moles was pronounced dead in 1985 in District 13 of either Oklahoma or Ohio (no one can remember). Soon after, he was brought back to life, and he has recovered miraculously, having sustained only major, permanent, lifelong side effects. Even though his condition was caused by infection, he maintains that vaccines are the reason for his autism. Certain inconsistencies, such as the fact that he never received any vaccines and is not actually autistic, don't bother him in the slightest bit. Mr. Moles is *not* a smokey, chocolatey Mexican sauce *or* equal to 6.0221415×10^{23}. He may, however, be a member of the Talpidae family.

Josh first stumbled upon video games when he tripped over a Christmas present his mother had put by the tree. Inspired by these video games, he would often dress up in a trenchcoat and invite his "friends" over for some "Dark Fighting". His claim to fame came in 2005, when he won the XBOX "UNO" tournament. He unfortunately never realized that the word "UNO" actually referred to the total number of players in the game. It did, however, inspire him to create a "2nd person shooter" game.

While most people like to socialize with others, Josh does not engage in this activity, claiming it "makes about as much sense as watching 4 Kirby's playing Super Smash Bros." Instead, he frequently goes to malls wearing small running shorts and tells everyone to look at his white thighs or "mayonnaise jars", as he likes to call them. He is fluent in moustache, chef, Intercal, Malbolge, Piet, and lolcode.

Josh has three children, all of which have had amber alerts issued by their real parents. When he isn't in prison, Josh likes to paint pictures of himself in prison. He also volunteers at the local hospital in the Dermatology department because he heard people go there to "check out Moles.

Erik Rhodes Fluent in Lorem Ipsum and cat stroking, Erik “Iced Source” Rhodes has published threes of papers in his lifetime and more often than not is able to perform this feat this while wearing only one shoe. Perpetually confusing those around him, he enjoys cheese bowling, tactical sneezing, and sand juggling. He is currently pursuing his goal of being called “sir” without subsequently being told, “you’re making a scene”. He is not Batman.

Erik received national fame upon the discovery of the “Iced Source”, but shortly thereafter several anonymous tips came surfaced and it was confirmed that this was in fact simply an anagram of “Code Cruise”. A jack of some trades and a master of pun, Erik was not about to let this career-destroying fiasco obliterate any more than his personal, professional, and family life. Soon thereafter, he returned to school to pursue an unprecedented triple-associates in speed-yodeling, industrial banjo, and Jimmy-legging.

Elected the honorary sub-mayor of a remote Alaskan village, Erik quickly fled the frozen tundra after being incapable of parallel parking his dogsled in the designated location, even after trying for several months. This event served only to temper his persistence and to this day, he regularly attempts to occasionally sit mostly still for part of the day, but has been unable to complete this goal at the time of this publication.

He just doesn’t understand avocados, and discovered his interest in parallel computing in the late 2000’s upon discovering that the dial-up sound was more than a “phat beat”. Since then, he has been working on parallel embarrassing pun algorithms and is investigating a bleeding-edge field known as ”massively shame” parallelism. When questioned, leading researchers in the field have referred to his latest works as “that doesn’t even make sense”, and “please leave me alone”.

Fig. 1: Quicksort Algorithm[5]

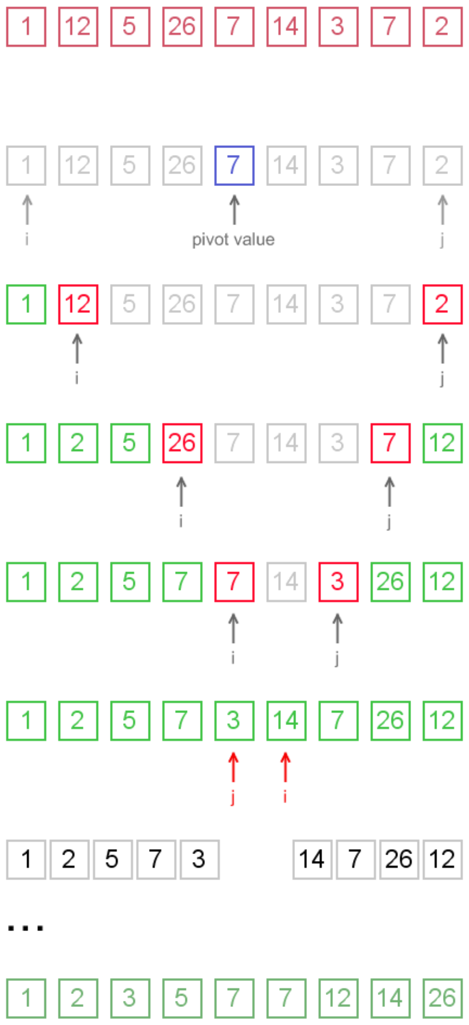


Fig. 2: Domain Decomposition

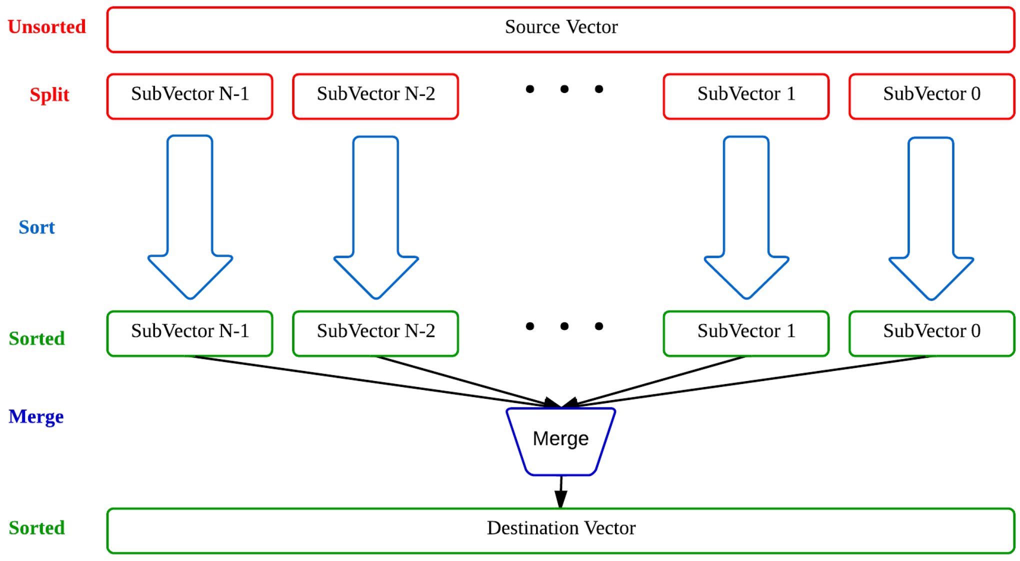


Fig. 3: MMS threading model

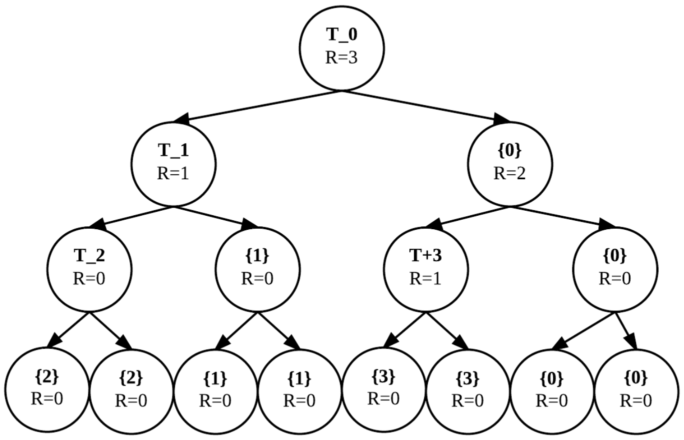
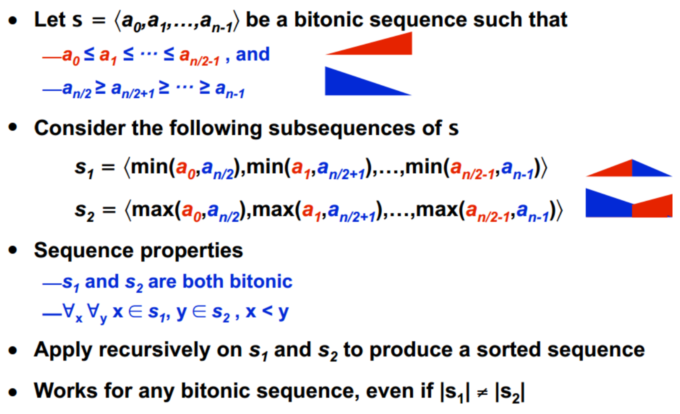


Fig. 4: Bitonic Split [6]



- Let s = $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ be a bitonic sequence such that
  - —$a_0 \leq a_1 \leq \cdots \leq a_{n/2-1}$, and
  - —$a_{n/2} \geq a_{n/2+1} \geq \cdots \geq a_{n-1}$
- Consider the following subsequences of s
  - $s_1 = \langle \min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}), \ldots, \min(a_{n/2-1}, a_{n-1}) \rangle$
  - $s_2 = \langle \max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}), \ldots, \max(a_{n/2-1}, a_{n-1}) \rangle$
- Sequence properties
  - —s_1 and s_2 are both bitonic
  - —$\forall_x \forall_y\ x \in s_1, y \in s_2 , x < y$
- Apply recursively on s_1 and s_2 to produce a sorted sequence
- Works for any bitonic sequence, even if $|s_1| \neq |s_2|$

Fig. 5: Bitonic Build [6]

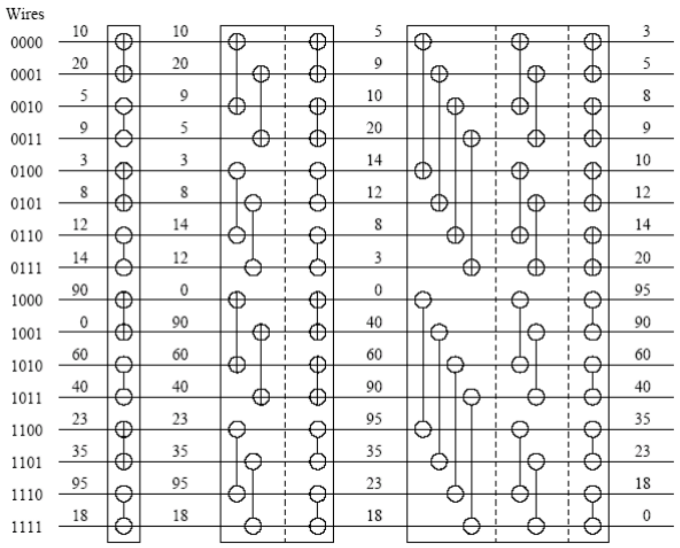
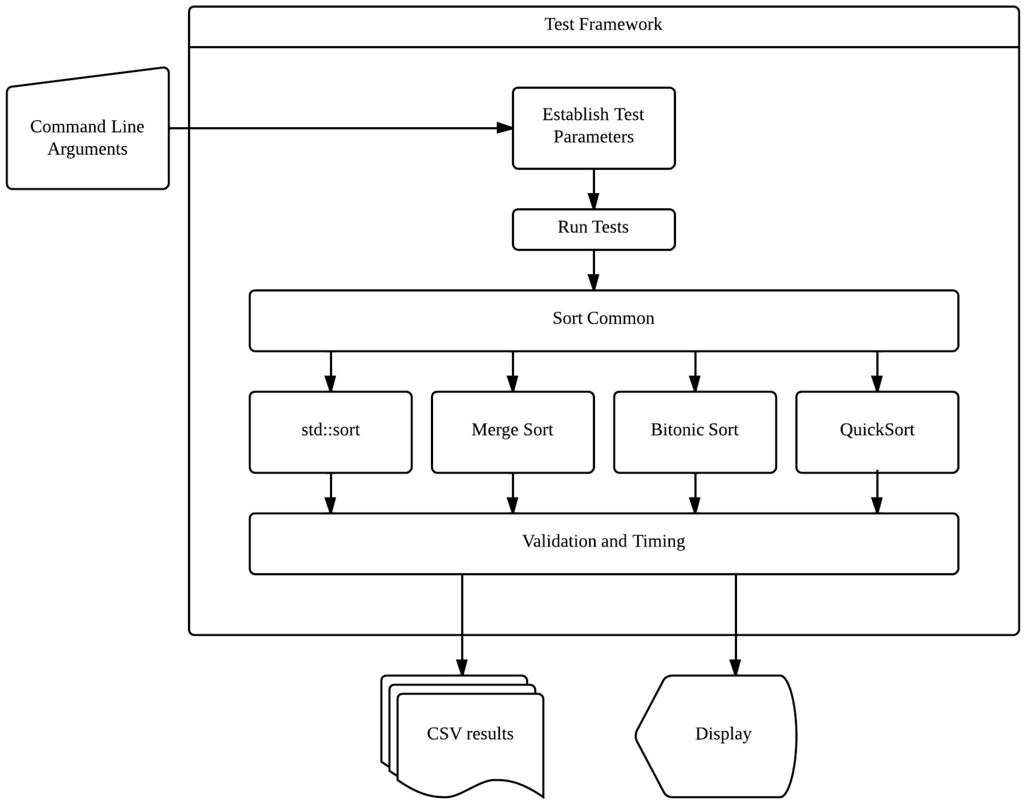


Fig. 6: Framework



Listing 1: Valgrind Output

```
$ valgrind --leak-check=full ./laser-patroll
<removed>
HEAP SUMMARY:
    in use at exit: 56 bytes in 1 blocks
  total heap usage: 267 allocs, 266 frees, 18,839 bytes allocated

LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 56 bytes in 1 blocks
        suppressed: 0 bytes in 0 blocks
Reachable blocks (those to which a pointer was found) are not shown.
To see them, rerun with: --leak-check=full --show-reachable=yes

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

Fig. 7: Time vs Threads Large Data (Size = 134217728)

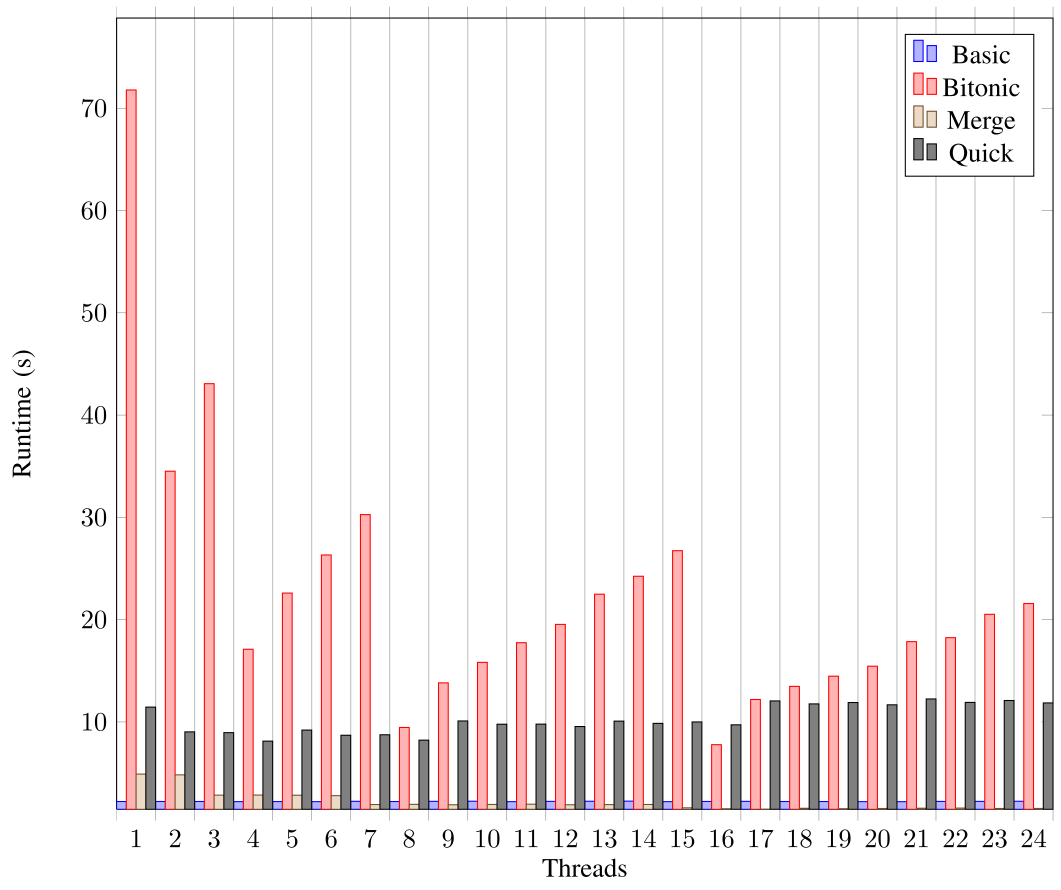
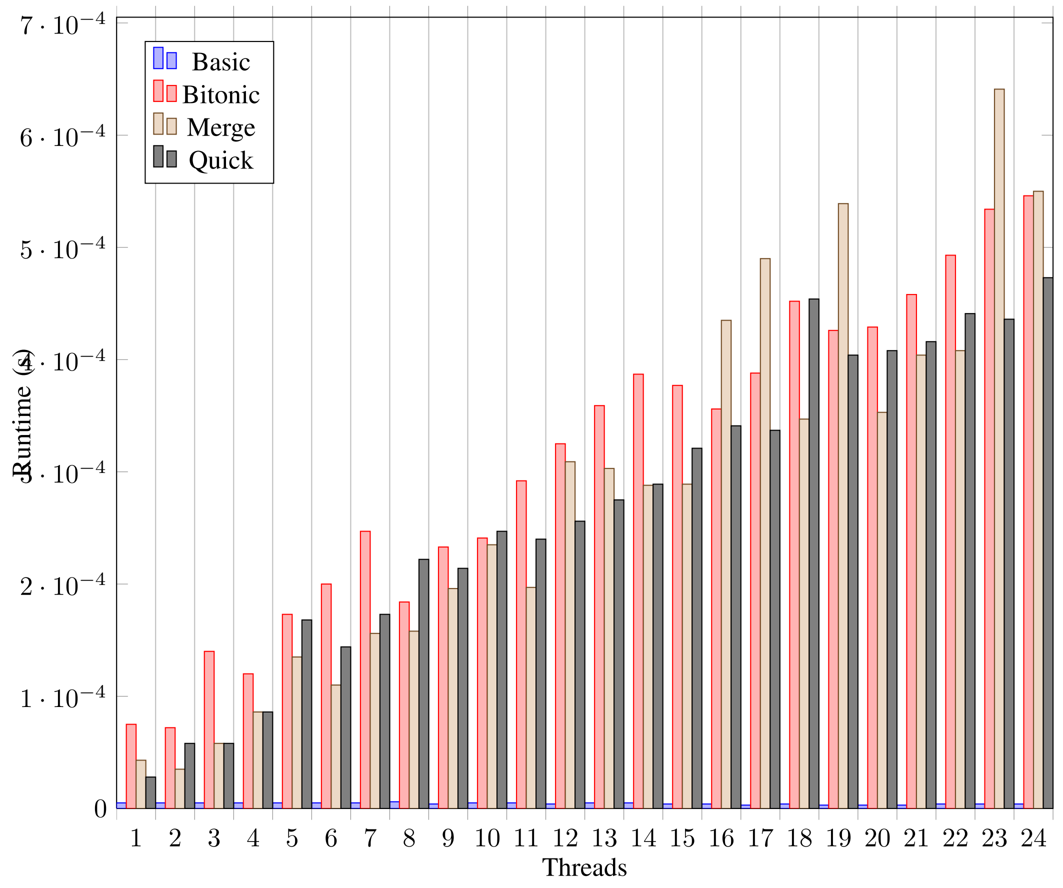


Fig. 8: Time vs Threads Large Data (Size = 512)


Basic
Bitonic
Merge
Quick
Runtime (s)
Threads
$7 \cdot 10^{-4}$
$6 \cdot 10^{-4}$
$5 \cdot 10^{-4}$
$4 \cdot 10^{-4}$
$3 \cdot 10^{-4}$
$2 \cdot 10^{-4}$
$1 \cdot 10^{-4}$